

# BEYOND CORE: MAKING PARALLEL COMPUTER I/O PRACTICAL \*

DAVID WOMBLE<sup>†</sup>, DAVID GREENBERG<sup>†</sup>, STEPHEN WHEAT<sup>†</sup>, AND ROLF RIESEN<sup>‡</sup>

(Proceedings of the Dartmouth Institute for Advanced Graduate Studies, 1993.)

**Abstract.** The solution of Grand Challenge Problems will require computations which are too large to fit in the memories of even the largest machines. Inevitably, new designs of I/O systems will be necessary to support them. Through our implementations of an out-of-core LU factorization we have learned several important lessons about what I/O systems should be like. In particular we believe that the I/O system must provide the programmer with the ability to explicitly manage storage. One method of doing so is to have a partitioned secondary storage in which each processor owns a logical disk. Along with operating system enhancements which allow overheads such as buffer copying to be avoided, this sort of I/O system meets the needs of high performance computing.

**1. Introduction.** The solution of Grand Challenge Problems will require computations which are too large to fit in the memories of even the largest machines. The speed of individual processors is growing too fast to be matched with increased memory size economically. Successful high performance programs will have to be designed to run in the presence of a memory hierarchy. Great efforts have already been made to optimize computations for the fastest end of the hierarchy – the use of high speed registers and caches. The result has been the creation of optimized codes such as those in the BLAS. At least as large an effort must now be made to address the slow end of the hierarchy – the use of secondary storage such as disks or even SRAM backing stores.

Traditionally the maintenance of this end of the memory hierarchy has been lumped under the category of I/O and left to the control of the operating system. We will argue that more explicit management of disk I/O is necessary for high performance. The application will have to provide, at the minimum, directives which allow the OS to optimize disk access and may have to provide explicit instructions as to how disk I/O is to be performed. As with the case of the BLAS, we expect that a lot of the explicit management will be packaged in libraries.

The purpose of this paper is to describe our experiences in producing high performance codes that act on data sets which are too large to fit in core memory. Some of the lessons we have learned are algorithmic (ways to structure the code to reduce the I/O bottleneck) while others are systems related (what features of the OS can make producing high performance codes easier). Throughout this report we will use LU factorization (a common scientific kernel routine which is fairly prototypical) as a running example.

**2. Characteristics of I/O.** The I/O requirements of a program can be divided into three phases: initial input of data, maintenance of temporary memory, and the output of results.

The transfer of data into the machine can be from HiPPI, disk or from another computer, and is a one-time operation. Usually the limiting factor is the speed of the hardware. If the initial loading of data is done improperly, it can add significant overhead to a computation, but it is rarely an insurmountable bottleneck. One important issue is the format of the data. Because the input to one program or routine is often the output of another, it is not always possible for a program to input the data in the most efficient format. The translation of data from one format (eg. row major order) into another (eg. block column order) can be of great importance. We will have more to say about such translations later in this report but otherwise will not further consider the input of data.

The output of results is similar in character to the input of data. A conversion of format may be necessary but each item is written out just once. Typically an acceptable method of streaming the output

---

\* This work was supported by the U.S. Department of Energy and was performed at Sandia National Laboratories operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789

<sup>†</sup> Sandia National Laboratories, Albuquerque, NM, 87185

<sup>‡</sup> University of New Mexico, Albuquerque, NM, 87131

can be found. In the case of repeated output of intermediate results (eg. graphic representation of a physical simulation) the output requirements remain static throughout.

The handling of temporary values is much more problematic. Temporary values must be both written and read. The order in which they are accessed can change over time. In addition the critical path of the computation will rely on data being present in fast memory. If the management of temporary memory is not efficient, it can slow the whole computation to a crawl. Virtual memory is one possible approach to maintaining temporary storage which is being provided by vendors. While this simplifies the programming, no virtual memory system can perform as well as a code written by a programmer who understands the algorithm being implemented; the overhead of a virtual memory system often defeats the advantage of using a parallel supercomputer, i.e., computational speed. In the rest of this report we will explore what efforts are required by the programmer to produce acceptable management of temporaries. Along the way we will discuss what the operating system can provide to make the programmer's job easier.

**3. Partitioned Secondary Storage.** Our goal is to have high performance on large numbers of processors. We cannot afford to pay the overhead for general purpose I/O service. Instead we must understand the characteristics of our problems and tailor the I/O system to our problems. Similar issues have arisen concerning the partitioning of data amongst processors. We will use the lessons learned there as a guide.

For the large scientific codes written at Sandia, it has become apparent that the overhead of shared memory emulation is often large. A message passing paradigm is preferred because of its higher performance. In our experience, the key aspect of this paradigm is that the programmer explicitly arranges for data which is used locally to be stored in local memory. We would like the I/O system to maintain this view of local storage.

Our solution is for each processor to have its own logical disk. The data on a processor's disk will be treated similarly to the data in its local memory: the processor will have sole control of this data. Any sharing with other processors will be through explicit message passing. We call the strategy of having each processor own a logical disk partitioned secondary storage (PSS).

PSS allows the application to control data locality. The programmer always knows where the data is and can therefore reliably plan the overlap of computation and I/O and message passing. This control of the data meshes well with the message passing paradigm. If, as is often the case, the program has been parallelized by creating processes which work mostly on local data, we do not want the I/O system to destroy the locality in the search for general parallelism. Using PSS the program can still be divided up so that the compute work is evenly balanced among the processors and so that data can be reused as often as possible. This data reuse/locality is critical for good performance. PSS removes the impact of limits on local memory size by allowing the computation to be decomposed so that each process can be designed as if it had access to a large memory.

We remark here that the programming required to make effective use of PSS is more complicated than that required for shared memory emulation or for a virtual memory system. However, because PSS strictly adheres to the distributed memory paradigm, we expect that anyone programming a distributed memory machine using explicit message passing will be able to use PSS easily and effectively.

The requirement that each processor has its own logical disk does not necessitate that each processor has its own physical disk. In the near future it is possible that there will be a small disk associated with each processor board. In the mean time it is allowable in our model (and sometimes desirable) for the OS to simulate a virtual disk for each processor using some smaller number of physical disks. The important point is that each processor acts as if it has its own disk. When each processor actually has its own disk, very high performance will result. When the processor's disks are virtual, there will be some loss in performance. Even so, the OS should be able to provide a fairly high speed emulation. If the I/O for each virtual disk is optimized by the application, then the OS need only manage the interleaving of the virtual requests onto the physical device.

One additional advantage of allowing virtual disks is that the OS can take advantage of device paral-

lelism. Systems such as RAID devices can use parallel accesses to slow disks to simulate a faster disk and provide redundancy for fault tolerance. This sort of parallel I/O is quite valuable, but it is orthogonal to the issues of parallel I/O that we are discussing.

There are several alternatives to the PSS paradigm. Each has its advantages and areas of applicability, but we believe that none is as effective for large scale scientific computing as PSS. One such alternative is the shared parallel file system (PFS) in which a single file is shared by all processors and is spread across many logical disks. This is typically a higher level approach than the PSS and requires more bookkeeping by the operating system. One advantage is that the format of data on the disks is transparent to the programmer. The programmer cannot explicitly control the placement of data and is thus absolved of the need to spend effort tuning it. A second advantage is that it is possible to share data between processors through the file system. Since all processors share one file it is possible for one processor to read what another wrote. The disadvantage is, of course, a reduction in performance. The overhead necessary to synchronize access can be very high. In addition, since a piece of data may be later required by any processor, there is no way for the I/O system to know ahead of time a good place to store it.

Of course there has been a great deal of research on how to maintain virtual memory both in single processors and in PFSs. Caching strategies and selective duplication of data with explicit invalidation and many other approaches have been developed to provide general purpose virtual memory. Some of these methods work very well for certain types of programs. We contend, however, that they are unlikely to compete with strategies developed specifically for particular scientific programs. Some systems, such as Mach, allow the user to supply specific routines for use by the virtual memory/PFS system. In this case we suspect that users will choose to implement a PSS type system, which by-passes the general mechanisms and returns the ability to control locality to the application.

**4. Systems Implementations.** We do not intend to recommend any particular implementation of an I/O system. So far we have merely argued for the inclusion of a high performance option that looks like a PSS. We have, however, been using a particular OS called PUMA [7]. PUMA was designed by Sandia National Laboratories and the University of New Mexico initially for the nCUBE, and there are plans to port it to the Intel Paragon. Because PUMA is our own operating system we have been able to incorporate several features which we have found particularly useful.

PUMA supplies the user with the ability to ask for a “block server” type of I/O. Rather than having a file with all the associated overhead of opening, closing and maintaining handlers, the user is provided with disk access routines that parallel those of memory (e.g., disk can be allocated using the `dalloc()` command). Routines are also provided to transfer blocks of data between disk and memory. In the case of our LU factorization program, we have sustained near maximum transfer rates and reduced the nonoverlapped I/O time for an  $n$  element matrix to  $O(n^2)$ . This is discussed further in the next section.

A potential pitfall in memory management is excessive system buffering. Because of the large volume of data that can be transferred, extra system buffering can soak up both time and memory. The cost of buffering in both I/O and message passing is well known [1]. PUMA supplies the user with the ability to create *portals*. A portal is a section of memory for which the user makes a “contract” with the OS. The contract specifies ways in which the system and the user can share access to the section of memory. When both sides obey the rules, it allows the system to read and write I/O messages directly into user space. No system buffers or memory-to-memory copies are necessary. Since the PSS paradigm is such a simple use of disk, it is easy to produce a contract which allows the system to very efficiently perform the required I/O.

**5. Algorithmic issues in out-of-core programming.** The primary means by which an applications programmer can extract performance from an I/O subsystem is by algorithmic modifications. For example, the programmer may be able to change the data storage format on the disk to give a more efficient access pattern or combine subproblems normally performed sequentially to reduce the total amount of I/O required. We examine several algorithmic issues below using LU factorization as the sample problem.

**5.1. Data decomposition.** The first and most obvious task in converting a program to run using a disk is to decide which subset of the input data or intermediate calculations should be in core at any given time. For example, when computing the  $n \times n$  matrix product  $C = AB$ , the optimal I/O complexity is achieved by writing the product in terms of the four  $n/2 \times n/2$  matrix product as follows.

$$\begin{aligned}C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}\end{aligned}$$

where the subscripts designate the location of the submatrices. If the submatrices are too large to fit in memory, the matrix is recursively subdivided [8].

A similar procedure can be used to recursively subdivide LU factorization without pivoting [4]. If pivoting is necessary, however, entire columns must be held in memory. A recursive procedure can also be designed in this case (which we discuss below in the section on I/O complexity), but the more natural decomposition is to treat blocks of columns as the basic I/O unit. This leads to a slightly less efficient, but simpler algorithm. Another decomposition is to store a collection of rows and columns from the left and top sides of the matrix in memory at once. While this leads to a different ordering of computations, the I/O complexity is the same as the column oriented approach.

Thus there are several competing factors affecting data decomposition. On the one hand, there is the desire to reduce the amount of I/O required. On the other hand, there is the desire for ease of programming. In some cases it will be worth sacrificing a little in I/O time if it allows for simpler code. However, the system should provide enough flexibility so that the programmer can choose the appropriate trade off.

**5.2. Data format.** Once the data decomposition has been chosen, the format of data on the disk must be chosen. For example, if a matrix is needed by rows, it does not make sense to store it on disk by columns. Similarly, if a matrix is needed in blocks, as in the recursive algorithms described above, it does not make sense to store it by either rows or columns. Several reports have examined this issue and found that pre- or post-permutations of data lead to a substantially reduced running time for many computations [3].

Sometimes a single computation will require data in different formats for different subcomputations. At other times, as noted above, the input or output interface may require data in a different format from that which is optimal for the computation. Thus the ability to convert between formats can be important. Several authors have noted that using the I/O system alone to perform these conversions can be quite expensive. Typically data which is contiguous in large blocks in one format is scattered in another. Since disk rates are much lower when servicing scattered small requests, the I/O for the scattered format will suffer. A better alternative is to always read and write data to and from the disks in the disks' current large block format. When data is needed in a new format, the interconnection network can be used to reorder the data, and then write to the disks in the new format (see Cormen [2]).

In our most efficient implementation of column-oriented LU factorization, we use two formats. The first is used for the unfactored blocks of the matrix where the matrix is stored by columns. The second is used to store intermediate results and the final (factored) matrix. In this the column blocks are permuted so that the lower triangular entries are stored contiguously and are followed by the upper triangular entries. Because the lower triangular portion must be read into memory repeatedly, this results in less bookkeeping by the program and larger blocks of data transferred.

**5.3. I/O complexity.** There has been quite a bit of work done on the I/O complexity of several problems, including permutation, sorting, FFT's, and matrix-matrix multiplies [2, 6, 8]. Most of the work has dealt with the shared memory model. The bound on the amount of I/O can be used in the distributed memory case if an I/O operation consists of all processors reading in their contribution to a data block si-

multaneously. These theoretical results can provide a guide in developing out-of-core algorithms; although, as we demonstrate below, “good” out-of-core algorithms do not necessarily achieve optimal I/O complexity.

A recursive approach to out-of-core matrix-matrix multiplication was stated above. The number of disk operations (I/O complexity) for this algorithm is  $O(n^3/B\sqrt{M})$ , where  $n$  is the size of the matrix,  $M$  is the size of memory, and  $B$  is the total size of one I/O request (across all processors) [2, 8]. This can be seen by observing that the complexity of combining the eight subproducts is  $O(n^2/B)$ , and that the matrix must be recursively subdivided  $k$  times, where  $k$  satisfies  $n/2^k < \sqrt{M}$ .

We can use the same technique to derive an upper bound for LU factorization,  $\text{LU}(A)$  (without or without pivoting). As before, we subdivide the matrices  $A$ ,  $L$  and  $U$  into four submatrices, designated by subscripts. We also designate the LU factorization algorithm by  $[L, U] = \text{LU}(A)$ , where the matrices  $L$  and  $U$  are the resulting matrices. We can now write the LU factorization algorithm  $\text{LU}(A)$  recursively as follows.

$$\begin{aligned} [L_{1,1}, U_{1,1}] &= \text{LU}(A_{1,1}) \\ U_{1,2} &= L_{1,1}^{-1}A_{1,2} \\ L_{1,2} &= 0 \\ U_{2,1} &= 0 \\ L_{2,1} &= A_{2,1}U_{1,1}^{-1} \\ [L_{2,2}, U_{2,2}] &= \text{LU}(A_{2,2} - L_{2,1}U_{1,2}). \end{aligned}$$

Now, because the I/O complexities to compute the products  $L_{1,1}^{-1}A_{1,2}$ ,  $A_{2,1}U_{1,1}^{-1}$ , and  $L_{2,1}U_{1,2}$  are  $O(n^3/B\sqrt{M})$ , we can express the number of I/O operations to factor an  $n \times n$  matrix,  $T_{LU}(n)$ , by the recursion

$$\begin{aligned} T_{LU}(n) &= 2T_{LU}\left(\frac{n}{2}\right) + O\left(\frac{n^3}{B\sqrt{M}}\right) \\ &= O\left(\frac{n^3}{B\sqrt{M}}\right). \end{aligned}$$

It is interesting to note that pivoting does not change the upper bound. This can be explained by observing that the complexity of rearranging the rows of a matrix is  $O(n^2/B)$  [8], which is a lower order term. We also note that the upper bound for LU factorization is the same as that for matrix-matrix multiplication.

The recursive algorithm for LU factorization can be difficult to implement so we now describe a simpler algorithm and discuss its complexity. We begin by dividing the matrix  $A$  into  $b$  column blocks of size  $n \times k$ , where  $nk = O(M)$  and denoting these blocks by  $A_i$ ,  $i = 1, \dots, b$ . Denoting the corresponding components of  $L$  and  $U$  by  $L_i$  and  $U_i$ ,  $i = 1, \dots, b$ , we can write LU factorization as follows.

```

for  $i = 1, \dots, b$ 
  for  $j = 1, \dots, i - 1$ 
    update  $A_i$  with  $L_j$ 
  end for
  compute  $L_i$  and  $U_i$ 
end for.
```

The high order I/O terms in this algorithm arise from the repeated reading of the  $L_i$ . In particular  $L_i$  has approximately  $(nk - (i - 1/2)k^2)$  entries and must be read  $b - i$  times. Summing this up for  $i = 1, \dots, b$ , we see that  $O(n^4/M)$  entries must be read yielding an I/O complexity of  $O(n^4/MB)$ .

The complexity of the second algorithm differs from the theoretical upper bound by a factor of  $n/\sqrt{M}$ , which for large matrices may be substantial. This can be mitigated in practice by overlapping the I/O with

computation. In particular, we note that the read of  $L_{j+1}$  can be accomplished while  $L_j$  is being used for computations. This reduces the “visible” I/O complexity to  $O(n^2/B)$ .

**5.4. Numerical stability.** The primary concern of the programmer must be the stability and correctness of the algorithm being implemented. One of the greatest temptations is to change the order of the computations in an established algorithm, or even to modify the algorithm, to reduce the amount of I/O required. An example of the latter is the use of pivoting in LU factorization. In serial or in-core algorithms, partial pivoting by rows or columns is considered necessary for stability. Early versions of out-of-core LU factorization, however, have stored square blocks of data in-core and restricted the pivot searches to the portion of a column currently in core. There are several reasons for doing this: storing larger square blocks (instead of entire columns) in memory allows more efficient use of the BLAS 3 library and reduces the I/O by a small amount. The disadvantages are that the factorization may not be stable for a large class of problems, the use of the code can not be “black box,” and the results can depend on the number and configuration of the processors. Based on the results of the previous subsection that the I/O complexity of LU has the same upper bound with and without pivoting and that much of the I/O can be overlapped, we recommend that partial pivoting be included in all general LU codes.

**5.5. Using libraries.** Finally, we remark that the choice of data decomposition and data format are dictated by the desire to use standard libraries like the BLAS. It is not unusual to see a factor of 10 speedup for optimized library code over standard FORTRAN or C code. This is one reason for restricting pivot searches. In our implementation of the LU factorization, we have made many of our choices to keep contiguous columns of data.

It is our hope that I/O libraries can be developed based in part on this work and the work of others. However, any successful I/O library must support a wide range of data formats and permutation functions to avoid forcing the programmer to adopt less than optimal formats for computation.

**6. Implementation.** The test program for our work on I/O has been out-of-core LU factorization. We have implemented this algorithm as described in previous sections. Specifically we

- use the block server implementation of PSS now incorporated into the PUMA operating system,
- use a column-block decomposition of the matrix for disk I/O and overlap I/O and computations as described in the section on complexity (a torus-wrap decomposition of matrix elements to processors is used within the column blocks [5]),
- permute a column block after its LU factors have been computed to make the elements of the lower triangular portion contiguous within each processor’s partitioned storage, and
- use partial pivoting to ensure numerical stability.

We note that the similarity of the update code to the factorization code in the column-block LU factorization, together with the simplicity of using the PSS paradigm within the distributed memory environment, has enabled us to write an out-of-core LU factorization code that is very similar to the in-core code and required very little additional work.

The test machine for our work has been the nCUBE 2 at Sandia National Laboratories. This machine has 1,024 nodes, each with a processor capable of 2.1 double precision Mflops/second (using the BLAS library) and 4 MBytes of memory. The disk system consists of 16 disks, each with its own SCSI controller. Each disk holds one GByte of data. The operating system used for these runs was PUMA.

Ideally, we would present experiments that make use of the entire capacity of the machine. Unfortunately, due to the cubical growth of computation time, a single run of the largest matrices requires several hours of computer time. Therefore, we instead present two medium sized runs to demonstrate the ability to overlap I/O with computation and several small sized runs to highlight the dependency on available memory and on the number of processors used.

Table 1 shows the results of running our LU factorization algorithm for a  $10,000 \times 10,000$  matrix on 64 processors varying the amount of memory available to the algorithm. In the first run, each block of columns

could be made large enough to cover the matrix with 14 block, while in the second run, half the memory was not used thereby halving the potential size of a block and doubling the number of blocks necessary. The increase in the number of blocks almost doubled the amount of I/O done and significantly increased the total run time. The last column records the amount of time spent doing I/O that could not be overlapped with computation, which we note is almost constant as predicted by the complexity results.

TABLE 1  
*Times to factor a double precision  $10,000 \times 10,000$  matrix using 64 processors on the nCUBE 2*

number of column blocks	column-block size (bytes/proc)	memory used	total I/O (bytes)	total time (sec)	total I/O time (sec)
14	893,214	94%	4,990,728,192	7,226	85
27	463,148	49%	8,612,713,472	10,733	87

The increase in total time is almost entirely due to increased interprocessor communication. The amount of communication required is  $O(n^2 b \sqrt{p})$ , where  $b$  is the number of column blocks, and  $p$  is the number of processors. Thus the memory size is important in that it defines the grain size for the computation, but not because it affects the amount of visible I/O.

Table 2 shows the dependence of the total run time and total I/O on both the number of processors and the memory used for a  $2,048 \times 2,048$  matrix. The non-overlapped I/O time is not shown because the small size of the matrix allowed effective caching of data by the disk software in some cases. (Again, the small size was chosen to enable us to do a larger number of runs.) The results again show that the total I/O is inversely proportional to the amount of memory available to the program. The increase in total time, however, is the result of increased interprocessor communication. The data in Table 2 does show that the total I/O is almost independent of the number of processors. Thus the algorithm scales well to large numbers of processors.

TABLE 2  
*Times to factor a double precision  $2,048 \times 2,048$  matrix on the nCUBE 2*

p	number of column blocks	column block size (bytes/proc)	memory used	total I/O (bytes)	total time (sec)
16	4	524,800	55%	71,417,856	231
16	8	242,400	26%	131,866,624	254
32	2	525,312	55%	33,619,968	117
32	4	262,656	28%	71,467,008	126
32	8	131,328	14%	131,923,968	141
64	2	262,656	28%	33,619,968	67
64	4	131,328	14%	71,532,544	77
64	8	65,664	7%	132,136,960	99

**7. Conclusions.** In this paper we have discussed several aspects of “out-of-core” programming on parallel machines. This is part of the larger problem of moving data into and out of these computers and is characterized by repeated reformatting and transfer of data to and from secondary storage. In particular, we discussed the advantages and disadvantages of several paradigms for disk usage and made the case that an efficient partitioned secondary storage (PSS) is necessary for high performance scientific computation.

We also discussed many of the programming issues which arise when using a disk system for temporary storage. These included data partitioning, data format, I/O complexity, numerical stability and the use of libraries. In each of these discussions, we used LU factorization as an example and in the end showed that the use of PSS (the “block server” facility in the PUMA operating system) leads to a very efficient out-of-

core LU code. We also showed that an important feature of any I/O library or PSS system is the ability to do background I/O. This enabled us to substantially reduce the “visible” I/O time in LU factorization.

Even though we used an LU factorization code to demonstrate the ideas of parallel I/O, most of the discussions in this paper apply equally well to any code that requires repeated access to large blocks of data. (The discussion of I/O complexity, of course, is specific to LU.)

**Acknowledgments.** We would like to thank Tom Cormen of Dartmouth College for helpful discussions about I/O complexity and Barney MacCabe of the University of New Mexico for suggestions on many of the operating systems issues discussed.

## REFERENCES

- [1] C. M. BURNS, R. H. KUHN, AND E. J. WERME, *Low copy message passing on the alliant CAMPUS/800*, in Proceedings of Supercomputer '92, 1992, pp. 760–769.
- [2] T. H. CORMEN, *Virtual Memory for Data-Parallel Computing*, PhD thesis, MIT, 1993.
- [3] J. M. DEL ROSARIO, R. BORDAWEKAR, AND A. CHOUDHARY, *Improving parallel I/O performance using a two-phase access strategy*, Tech. Report SCCS-406, Northeast Parallel Architectures Center, 1993.
- [4] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, Johns Hopkins University Press, 2nd ed., 1989.
- [5] B. A. HENDRICKSON AND D. E. WOMBLE, *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Sci. Stat. Comput., (to appear).
- [6] J.-W. HONG AND H. T. KUNG, *I/O complexity: The red-blue pebble game*, in Proceedings of the Symposium on the Theory of Computing, 1981, pp. 326–332.
- [7] A. B. MACCABE AND S. R. WHEAT, *Message passing in SUNMOS*. Overview of the Sandia/University of New Mexico OS, now called PUMA., Jan. 1993.
- [8] J. S. VITTER AND E. A. M. SHRIVER, *Algorithms for parallel memory I: Two-level memories*, Tech. Report CS-92-04, Brown University, 1992.